Lessons from the PL.8 Compiler


Copyright 2012, Keith D. Cooper & Linda Torczon, all rights reserved.
Students enrolled in Comp 512 at Rice University have explicit permission to make copies of these materials for their personal use.
Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.

The Setting

First RISC Compiler
- Original target was IBM 801 minicomputer
- Tight coupling of architecture & compiler
- Later targets included S/370, MC680x0, & others
- Basis for XL compiler series for RS/6000

Research compiler
- Compilation speed was not critical
- Emphasis on code quality, methodology, & theory
- Several breakthrough ideas
- Underlying philosophy governs RISC compilers today
The Language

A PL/I Subset (80/20 rule)

• Threw out ON conditions (exception handling)
• Permanently enabled subscript range checking
• Replaced unrestricted pointers with offsets & areas
• Bit string lengths fixed and restricted
• New declarations for call-by-value & no internal static variables
• Relaxed implicit conversion rules
• Simplified rules governing arithmetic precision

Other front ends (eventually)

• Pascal, Fortran, & C

Compiler Summary

Intermediate Representation

• Linear, low-level, abstract machine code
• Byte addressable storage
• Unlimited set of symbolic or virtual registers
• High-level operations to encapsulate control flow

Optimization

• Use global methods (whole procedure)
• Expose every detail to uniform optimization

Structure

Translation → Optimization → Register Allocation → Final Assembly

MAX
MIN
MVCL
CHECK
**Principles**

**Assumptions**
- Register allocator does a great job (separation of concerns)
- Little or no special case analysis
- Broad set of optimizations covers the IR

**Doctrine**
- Data-flow analysis pays off, so do it when needed
- Passes are independent but complementary
- Code is shaped for optimization
- Optimize, elaborate, optimize
- Finite machine is the allocator’s problem

As a matter of timing, PL/8 came out at a time when DF analysis was well developed & before SSA was invented.

As in FORTRAN H, BLISS 11, and other classic optimizers

---

**Translation Phase**

**Simple front end**
- LALR(1) parser
- Bottom-up generation of IR
- No significant analysis during translation
- Some machine-specific detail creeps in (branch ranges)
- Shape the code for optimization (syntactic & local)

**Front end does not**
- Build a control-flow graph
- Analyze the content for special cases
- Pre-assign registers (other than the ARP)
**Optimizer**

**Structure**

- Many passes
  - Independent & interdependent
  - Single point of control
  - Repeats some passes multiple times
- IR is definitive representation
  - Re-derive rather than update
- Insert & eliminate rather than replace
  - Rely on dead code elimination

**Register Allocator**

**Graph coloring allocator** (see Chapter 13, EaC)

- Constructs precise interference graph
- Use interference graph for coalescing copies (unlike Chow)
- Machine-specific constraints modeled in graph
- Use smallest degree last coloring
- Allocator handles all spill decisions

Effectiveness - compiling the compiler

- For S/370 (16 gprs): little more than 50% of values spill
- For 801 (32 gprs): over 95% do not spill
- Coloring works better with larger register sets (spill heuristic)
Scheduling & Final Assembly

Schedule Twice
- Pre-allocation scheduling to avoid constraints
- Post-allocation scheduling to place spill code

Final assembly
- Convert allocated, scheduled IR to object code
- Two passes with some local fix-up (peephole)
- Generate debugging information, tags for link-time checking
- Added tailored procedure prologs and epilogs

Miscellany

Range checking
- One goal was to decrease overhead of checking
- Lots of intellectual effort invested in this problem (V. Markstein)
- Area + offset could be checked, pointer could not
- Cocke & P. Markstein report 5% to 10% overhead
  - V. Markstein reports (eventually) getting that down to 2%

Reliability
- PL.8 was built with PL.8
- Daily use improved actual & perceived reliability
### Results (from Cocke & Markstein)

<table>
<thead>
<tr>
<th>Transformation</th>
<th>Optimization Level</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-1</td>
</tr>
<tr>
<td>Dead code elimination</td>
<td>X</td>
</tr>
<tr>
<td>Value numbering</td>
<td>X</td>
</tr>
<tr>
<td>Local constant propagation</td>
<td>X</td>
</tr>
<tr>
<td>Global commoning, code motion</td>
<td>X</td>
</tr>
<tr>
<td>Dead code elimination</td>
<td>X</td>
</tr>
<tr>
<td>Strength reduction</td>
<td>X</td>
</tr>
<tr>
<td>Macro expansion</td>
<td>X</td>
</tr>
<tr>
<td>Dead code elimination</td>
<td>X</td>
</tr>
<tr>
<td>Value numbering</td>
<td>X</td>
</tr>
<tr>
<td>Local constant propagation</td>
<td>X</td>
</tr>
<tr>
<td>Global commoning, code motion</td>
<td>X</td>
</tr>
<tr>
<td>Register allocation (k = r-4)</td>
<td>X</td>
</tr>
<tr>
<td>Register allocation (k = r+4)</td>
<td></td>
</tr>
</tbody>
</table>

### Compiler option flags

- USEDEF
- Puzzle
- IPOO
- Heapsort
- Spill code

### Program

<table>
<thead>
<tr>
<th>Program</th>
<th>Compile time</th>
<th>Code Space</th>
<th>Run time</th>
<th>Compile time</th>
<th>Code Space</th>
<th>Run time</th>
<th>Compile time</th>
<th>Code Space</th>
<th>Run time</th>
<th>Compile time</th>
<th>Code Space</th>
<th>Run time</th>
<th>Compile time</th>
<th>Code Space</th>
<th>Run time</th>
</tr>
</thead>
<tbody>
<tr>
<td>USEDEF</td>
<td>19.7</td>
<td>12,138</td>
<td>0.720</td>
<td>19.7</td>
<td>5,386</td>
<td>0.230</td>
<td>31.7</td>
<td>6,390</td>
<td>0.134</td>
<td>34.2</td>
<td>6,098</td>
<td>0.129</td>
<td>51.2</td>
<td>5,942</td>
<td>0.124</td>
</tr>
<tr>
<td>Puzzle</td>
<td>6.2</td>
<td>2,790</td>
<td>1.330</td>
<td>5.7</td>
<td>1,682</td>
<td>0.730</td>
<td>9.3</td>
<td>1,778</td>
<td>0.670</td>
<td>10.2</td>
<td>1,782</td>
<td>0.670</td>
<td>14.7</td>
<td>1,698</td>
<td>0.620</td>
</tr>
<tr>
<td>IPOO</td>
<td>9.8</td>
<td>4,908</td>
<td>5.880</td>
<td>10.3</td>
<td>3,404</td>
<td>4.250</td>
<td>15.5</td>
<td>3,232</td>
<td>3.610</td>
<td>17.3</td>
<td>3,216</td>
<td>3.590</td>
<td>20.5</td>
<td>3,156</td>
<td>3.510</td>
</tr>
<tr>
<td>Heapsort (84 lines)</td>
<td>2.2</td>
<td>1,024</td>
<td>5,600</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heapsort (in PL/I)</td>
<td>0.83</td>
<td>432</td>
<td>2,260</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heapsort (in Fortran)</td>
<td>0.26</td>
<td>674</td>
<td>4,830</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### System 370, times in seconds

- USEDEF
- Puzzle
- IPOO
- Heapsort
- Spill code
**Notes on Results Slides**

- Level 0 pays for itself (smaller code)
- Global code motion & cse lengthen live ranges (level 0 to 1)
- Biggest payoff is level -1 to 0, then 0 to 1; global optimization compensates for longer lifetimes
- Level 3 only helps with spill code (made obsolete by Briggs)
- Spilling increases code space, but increased optimization makes up for it (zero wait state memory)
- USEDEF references complex data structures in nested loops
- Tests exclude reassociation; Cocke & Markstein report that reassociation removes up to 50% of the code in USEDEF’s inner loops; helps with spilling & speed
- No linear function test replacement
- Constant propagation underperformed expectations; initial values not represented in the IR
- Heapsort doesn’t show off Fortran H, because it doesn’t use the loop index variable as a subscript index!

**Key Points**

- Strong philosophical influence on later compilers
  - Single point of control
  - Repeat optimizations
  - Two-level IR
  - Separation of concerns (strong back end)
  - Reanalyze rather than update incrementally
- Scope of optimization
  - Notice large improvement from -1 to 0 \((local \ optimization)\)
  - Design emphasizes global analysis and optimization
  - Results show a payoff \((smaller \ than \ local, \ but, \ ...)\)
  - Contrasts with Fortran H’s emphasis on loop nests
- Hardware/software co-design
  - 801 ISA designed as target for this compiler
**Key Points**

Graph coloring register allocation
- Precise interference graph
- Uniform approach to spilling (no local RA)
- Powerful method for coalescing copies (vs. Chow)

Reassociation
- Recognized the potential & worked on the problem
- In the end, method did not work as promised

Triumph of global analysis & optimization
- Decade of new algorithms
- This compiler showed that, in practice, it all worked
- Did well against mature S/370 compilers (not just on 801)

**Common Threads**

Understanding the problems
- Fortran H — addressing patterns and loops
- PL.8 — range checking, pointers, exceptions, generality in PL/I

Limited transformation repertoire (did a few things well)
- Fortran H — five transformations
- PL.8 — ten transformations

Concentrate on high-payoff areas
- Fortran H — focus intensely on loops
- PL.8 — range checking, linkages, customization, & allocation